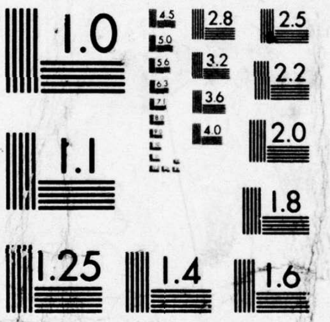
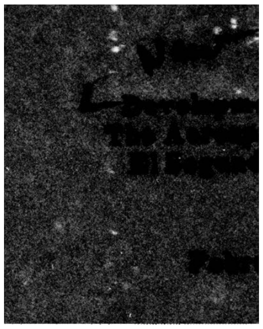
AD-A033 792 AEROSPACE CORP EL SEGUNDO CALIF DEVELOPMENT OPERATIONS F/G 9/2
MICROPROGRAM VERIFICATION AND VALIDATION.(U)
FEB 76 D J REIFER F04701-75-C-0076
UNCLASSIFIED TR-0076(6112)-4 SAMSO-TR-76-217 NL

| OF |
AD
A033792

END
DATE
FILMED
2-77


1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6


[illegible]

This interim report was submitted by The Aerospace Corporation, El Segundo, CA 90245, under Contract F04701-75-C-0076 with the Space and Missile Systems Organization, Deputy for Advanced Space Programs, P.O. Box 92960, Worldway Postal Center, Los Angeles, CA 90009. It was reviewed and approved for The Aerospace Corporation by R. O. Bock and G. P. Millburn, Development Operations. Capt. Edward V. Fanelli SAMSO/YAD, was the project officer.

Edward V. Fanelli JR.
Edward V. Fanelli, Capt, USAF
Project Officer
Computer Technology Division
Deputy for Advanced Space Programs

[illegible]
Frank P. Dyke, Lt Col, USAF
Chief, Computer Technology Division
Deputy for Advanced Space Programs

FOR THE COMMANDER

[illegible]
Floyd R. Stuart, Col, USAF
Deputy for Advanced Space Programs

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE — READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 18 19 SAMSO TR-76-217
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): 6 MICROPROGRAM VERIFICATION AND VALIDATION.
5. TYPE OF REPORT & PERIOD COVERED: 9 Interim rept. Jul~~y~~-Dec ~~1~~975,
6. PERFORMING ORG. REPORT NUMBER: 14 TR-~~00~~76(6112)-4

7. AUTHOR(s): 10 Donald J./Reifer
8. CONTRACT OR GRANT NUMBER(s): 15 F~~0~~47~~0~~1-75-C-~~0~~076

9. PERFORMING ORGANIZATION NAME AND ADDRESS
The Aerospace Corporation
El Segundo, California 90245

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Space and Missile Systems Organization
Air Force Systems Command
Los Angeles Air Force Station, CA 90045

12. REPORT DATE: 11 Feb~~ruary~~ ~~1~~976
13. NUMBER OF PAGES: 56

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 12 55p.
15. SECURITY CLASS. (of this report): Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Emulation
Microprogram
Simulation
Verification
Validation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Microprogramming has emerged as a central attribute of the small-to-medium-scale computers embedded within both military and industrial systems. Among the principal reasons contributing to the rapid growth of this technology and its wide application are that it is easy both to incorporate changes to a design and to emulate consistently another computer. Unfortunately, there exist many similarities between microprogramming technology as it exists today and the state of the art of software technology as it existed twenty years ago. Presently, microprograms are developed

DD FORM 1473 (FACSIMILE)

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409981 LB

19. KEY WORDS (Continued)

20. ABSTRACT (Continued)

by highly skilled personnel who are familiar with the inner workings of the target computer. They construct routines of microinstructions that are loaded into a read-only memory and are then tested. The test techniques employed are crude compared with those that exist for software. Yet, the complexity of the task in terms of timing and resource allocation of the microprogram in many instances exceeds that of most software programs written in an assembly or higher order language.

Considerable interest in developing improved methods for microprogram verification and validation has appeared because of the impact microprograms have on the reliability of the total computer system. The purpose of this report is to document results of recent investigations conducted to determine what the state of the art of microprogram verification and validation is and how it might be improved. Current methods and promising new techniques are reported and an assessment of the technology is made.

TABLE OF CONTENTS

PREFACE

The author wishes to thank Dr. H. Hecht and Capt. G. Sims for their encouragement and helpful criticisms. Acknowledgment is also made to B. J. Schmidt for her excellent efforts in preparing this report.

I. INTRODUCTION

The task of programming at the firmware level as distinguished from programming at the software level is made more difficult by the machine dependencies introduced when generating binary patterns to control data flow and sequencing at a detailed machine architectural level. Microprogrammers must possess superior capabilities to the average programmer because they must understand the details of the specific machine they are working with and its internal timing at the microarchitectural transfer level in order to be proficient at their jobs. They must be intimately familiar with the microinstruction repertoire and be concerned with optimizing their code because its repetitive execution dramatically impacts processor speed and its length impacts control memory size. Once the microcode is developed, microprogrammers must comprehensively test it to assure that it performs according to its specification. Unfortunately, tools and techniques to aid the microprogrammer in providing such satisfactory assurance are either still in development or in many cases totally lacking. It is envisioned that many of the test approaches developed for software can be adapted for use and applied to the problem of microprogram verification and validation. These should significantly reduce, but will not eliminate, the problem. Even if the state of the art of microprogram verification and validation were on a par with that of software testing, approaches would have to be developed to ensure that critical code (both software and firmware) satisfactorily performed as it was supposed to and did nothing more.

The purpose of this report is to document results of recent investigations aimed at quantifying the state of the art of microprogram verification and validation and to show how it can be improved. Methods currently being investigated can be categorized as follows:

a. Diagnostics. The collection of techniques used to monitor the execution of the microcode, determine if errors have occurred, and isolate code execution errors to code segments where faulty performance is indicated. Techniques being explored include probe insertion and dynamic monitoring.

b. Test and Evaluation. The collection of techniques used to monitor the execution of the microcode to determine whether it meets its performance and/or functional requirements (correct outputs for correct inputs). Techniques being explored include symbolic execution and the use of automated aids.

c. Program Proving. The collection of analytical techniques used to verify the correctness of the microprogram with respect to formally defined specifications of the mathematical intent of the program.

d. Simulation/Emulation. The collection of techniques used to observe the changes over time of either a dynamic model of the microprogram or the microprogram itself executing in some pseudo-operational environment.

e. Graph-Theoretic. The collection of mathematical techniques used to model the data-flow features of a microprogram represented as a directed graph to help analyze structural and timing delays.

f. <u>Monitors</u>. The collection of techniques used to monitor the execution of microcode and capture performance data without distorting timing. Techniques being explored include the use of hardware monitors, firmware probes, and content-addressable memories.

Current methods and promising new approaches discovered during our investigations are reported using this classification scheme.

II. BACKGROUND MATERIAL

A. DEFINITIONS

In order to eliminate confusion over what is meant by terminology, the following definitions were selected for use in this report.

1. DEBUGGING

The process of localizing and removing errors from a microprogram. Debugging starts with known errors and attempts corrections.

2. EMULATION

The ability of one system to execute machine language programs written for a different system (Ref. 1).

3. FIRMWARE

Microprograms resident in control store (Ref. 2).

4. MICROCYCLE

The cycle of control that performs the fetch and execution of a primitive microinstruction (Ref. 3).

5. MICROINSTRUCTION

A bit pattern normally stored in control memory that controls, at a primitive level, the processor hardware (Ref. 3).

6. MICRO-OPERATION

A primitive hardware operation, e.g., addition, shift, transfer into a register, etc.

7. MICROPROGRAM

A sequence of microinstructions written to mechanize on a computer a series of actions that achieve a desired result.

8. MICROPROGRAMMING

A technique for designing and implementing the control function of a data processing system as a sequence of control signals to interpret fixed or dynamically changeable processing functions. These control signals, organized on a word basis and stored in a fixed or dynamically changeable control memory, represent the states of the signals that control the flow of information between the executing functions and the orderly transition between these signal states (Ref. 1).

9. MICROPROGRAM VALIDATION

The test and evaluation of the complete microprogram aimed at ensuring that all functional and performance requirements specified have been properly implemented according to user-defined acceptance criteria that govern correctness (Ref. 4).

10. MICROPROGRAM VERIFICATION

The iterative process of determining whether the process of each step of microprogram development fulfills the requirements levied upon it by previous steps. For example, verification that an algorithm design has been satisfactorily implemented can be demonstrated by showing an acceptable correspondence between the computational results expected and those actually realized (Ref. 4).

11. TEST AND EVALUATION

The process of determining whether a microprogram will perform the functions for which it was designed in a satisfactory manner. Testing evaluates how well the specifications are met and helps detect errors in the implementation (Ref. 5).

B. MICROPROCESSORS AND MICROPROGRAMS

Microprocessors come in many different sizes and shapes and have many different microprogramming features. For convenience, available configurations can be discussed according to the classification scheme developed by Gilder (Ref. 6) with attention given to their microprogramming capabilities as follows:

1. MICROPROCESSOR CHIP SETS

Special-purpose integrated circuits with associated memories and electronics used to mechanize simple digital control and logic for specific applications (e.g., process controller for a gas turbine generator). Microprograms are small, non-changeable once installed, and transparent to the user.

2. MICROPROCESSOR-BASED LOGIC BOARDS

Special-purpose digital logic boards using microprocessor chips to interconnect its components and to interface logic to peripherals and memory for a variety of applications (e.g., teletype RS232 controller for interfacing data set to a computer). Microprograms are again small, non-changeable once installed, and transparent to the user.

3. GENERAL-PURPOSE MICROPROCESSORS

General-purpose, non-dedicated, user microprogrammable units are available from multiple vendors (i.e., Intel's 8080, Motorola's M6800, etc.). These units typically can address up to 65,000 bytes of memory and have sophisticated instruction sets. Microprogram languages and development systems (i.e., a simulator, an assembler or compiler, and a loader) exist and are used to help create microcode useful for a wide range of applications.

4. MICROPROGRAMMABLE MINICOMPUTERS

Minicomputers that implement their instruction sets and other system functions (i.e., recovery, etc.) in microcode and are user-microprogrammable exist (e.g., Varian 620, Nanodata QM-1, etc.) and are used for a variety of applications, including emulation of other machines. Microprograms developed for these machines are extremely complex because of timing and bit replication requirements.

We shall focus our attention on reviewing verification and validation techniques developed for general-purpose microprocessors and microprogrammable minicomputers with an emphasis on the latter throughout the remainder of this report.

C. MICROPROGRAM DEVELOPMENT CYCLE MODEL

The microprogram development cycle is simply a series of orderly, interrelated activities that lead to the successful completion of a set of microprograms. The periods of time during which these activities take place are called phases. A model of the microprogram development cycle containing seven phases is illustrated in Figure 1. Each of these phases is briefly described as follows:

1. REQUIREMENTS DERIVATION AND FUNCTIONAL ALLOCATION

The first activity in our model involves determining what the functional and performance requirements of the overall data system are and how to allocate them to various hardware, software, and firmware components. The necessary tradeoffs are conducted and alternatives evaluated to ascertain how best to achieve system goals. Provisions for quality are engineered into each system component during this phase, and the overall microprogram specifications and test plans are developed (the product of this activity).

2. MICROPROGRAM ANALYSIS AND DESIGN

The second activity undertaken involves generating technical solutions that are expected to satisfy the requirements specified and allocated to firmware. Analysis is conducted to provide an acceptable technical approach within the constraints imposed by the total data system environment. Acceptable solutions and

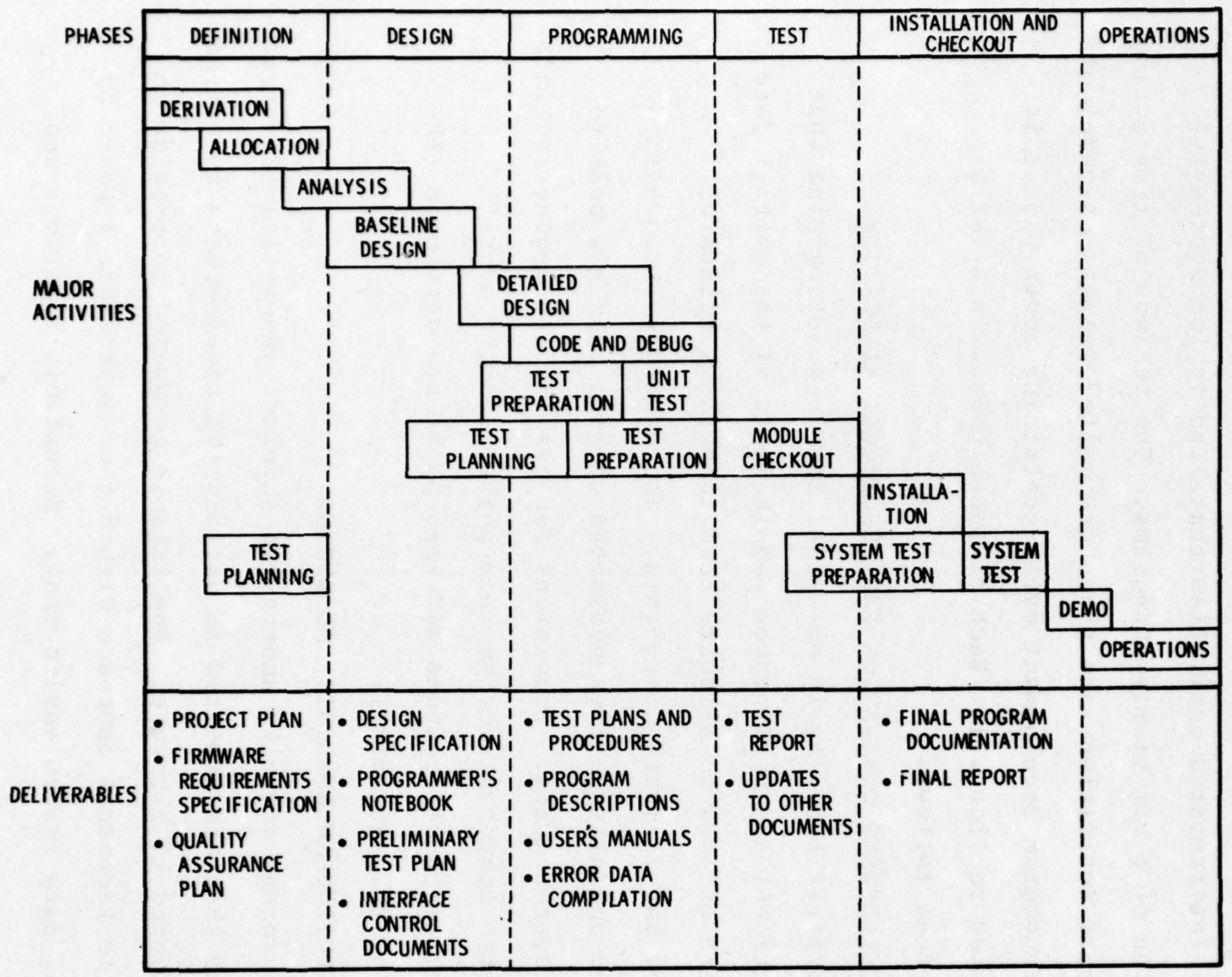


Figure 1. Microprogram Development Cycle Model

algorithms are devised and a code-to specification and a test plan published.

3. CODE AND DEBUG

The third activity involves production of executable microprograms by transforming the code-to specifications into sequences of microinstructions that are executable on the host computer. Debugging is usually accomplished on a simulator or emulator by the programmer, and executable microcode is the product.

4. UNIT CHECKOUT

The fourth activity involves confirming that the microprogram meets all of its design requirements and complies with applicable standards and conventions. Unit testing is accomplished by using simulated and/or real data to demonstrate each capability of the microprogram.

5. MODULE CHECKOUT

The fifth activity involves confirming that the integrated microprogram in its executable load module form (may contain several units) satisfies all functional, performance, and quality requirements specified for it and interfaces properly with both the hardware and software subsystems (usually modeled). Validation tests are conducted using simulated and/or real data to drive the subsystems through typical operational sequences.

6. INSTALLATION AND CHECKOUT

The sixth activity involves installing the tested executable load module into control memory and confirming that it operates satisfactorily. After the microprogram has been installed there is typically no chance to modify it without replacing the control memory. Bit-for-bit comparisons are run to ensure that the code loaded into control memory is the same as that which underwent checkout. Once the microprogram has been installed, integrated checkout is conducted to determine if the data system as implemented meets its requirements satisfactorily.

7. DOCUMENTATION

The final activity involves the publication of the documentation, which includes the user's manuals, program descriptions, error compilations, and programmer's notebooks. Exactly what documentation is required is a prerogative of the contracting agency and will vary from organization to organization.

The microprogram development cycle described above adheres to the bottom-up approach because of our desire to produce highly efficient microcode. The impact of a few wasted microseconds here and some control memory there is tremendous and must be controlled. We suggest that a top-down approach to microprogram development could be implemented profitably if the limitations and costs of current control memories were reduced and if higher order languages were developed for microcode development.

III. MICROPROGRAM VERIFICATION AND VALIDATION METHODS

A. DIAGNOSTICS

Diagnostics are the tools the microprogrammer uses to debug and check out his microprograms. They help him diagnose, isolate, and correct errors in the microcode by providing him with both static and run-time data gathered using conventional dump, snap, trace, and breakpoint techniques. Although several debugging and higher order language systems (Refs. 7,8,9) are available for use with general-purpose microprocessors, little attention has been paid by them to the question of providing adequate diagnostic support. Therefore, much of the material that follows recommends areas for microprogramming language designers to pursue instead of reporting on what exists.

Diagnostic tools can be classified as either static or dynamic (Ref. 10). Static tools operate at compile or assembly time, while dynamic tools operate at run time. In addition, there is also a class of tools that are language-based but are extrinsic to the language and its translator (Ref. 11). Each of these classes of tools will be briefly discussed in turn.

Static tools include translator diagnostics used to detect and report syntax and feature errors (i.e., proper nesting of loops, variable declarations, etc.), special debugging commands used to augment existing translator diagnostics and provide for interactive debugging, enforcer programs used to check whether the source code adheres to standards (i.e., module size, naming conventions, etc.), programs used to check the validity of interprogram communications and the passage of arguments, language statistics programs used to collect data on language feature usage (i.e., count of each program's statements by type), programs used to type check variables, programs used to make the source code more readable (i.e., indentation, etc), and an endless variety of other packages. Most of the programs used for static analysis were developed to automate the tedious, manual processes associated with desk-checking the program (Ref. 12). Therefore, their development for microprogramming may be justified in terms of dollar savings, and additional work is encouraged. A problem of more immediate concern is developing more meaningful diagnostics (i.e., error messages) that are easily understood and usable by a microprogrammer who by training is most likely an engineer and not a computer scientist.

Dynamic tools include execution profiles used to measure the program's performance by counting the number of times each statement is executed, trace histories used to provide timed records of statement executions and utilization of memory, monitor programs used to check user-defined assertions for range-checking, etc., traces used to display changes in the state of the system or of a variable as it occurs, dumps used to record information about the state of a computation at a given point in execution, snaps used to record intermediate data values during execution, and an endless variety of additional programs. Dynamic tools are used by the programmer to do his detective work and to comprehensively check his products. They provide him with meaningful data about the program's execution. Presently, few dynamic tools are available for microprogrammers. Those that are available were developed for general-purpose microprocessors and not for microprogrammable minicomputers. These tools allow the programmer to sit at a console and single-step through his program. He does this one instruction at a time to examine the status of the machine registers after each execution and determine whether they contain their expected values. Selective dumps can be requested to display all or part of the register data, and execution with substitutable parameters is available. Better and more powerful languages and debugging systems that recognize the parallelism associated with microprogramming could be made available to improve current technology. Many of the tools and techniques used for software can be transferred when standard languages are adopted for use in programming both general-purpose microprocessors and microprogrammable minicomputers. Emphasis should be placed on making the language translator do as much of the dog work as possible during the development (e.g., the translator is the natural place to instrument and monitor the source code).

The third type of language-based tools that exist is extrinsic to the language. Examples of such tools include packages used to produce symbol cross-reference tables, flow charts, etc. These packages are probably more economically handled by processors separate from the language system or by a preprocessor or postprocessor.

The three types of diagnostics discussed should not be considered independently. A good debugging environment consists of a variety of tools and a system for alternating among them as they are needed. An integrated system should be constructed that can be tailored to a user's needs. Interactive approaches with built-in safeguards that make the system resist mistakes should be pursued because a two-way conversational ability helps in the debugging process and increases productivity. Such systems when developed should allow the user to display data, do syntax checking, examine and modify register contents, provide selective traces during execution, measure program execution times, create hardcopy of specified program areas and memory locations, dump and modify memory, examine and set input/output, generate hardware interrupts, monitor significant variables, and set breakpoints and rollback points. Some of these capabilities exist in simulators developed for general-purpose microprocessors (e.g., Intel 4004/4040 Simulator). They should be extended and like systems developed for microprogrammable minicomputers. Systems such as DYDE (Ref. 13), EXDAMS (Ref. 14), and GRAPE (Ref. 15) offer examples of extended debugging systems developed for software that could serve as models for similar microprogramming developments. Although such systems could reduce errors, their proper use would be dependent upon the intelligence and skill of the microprogrammer using them.

B. TEST AND EVALUATION

Testing is the process of exercising a program to ascertain if it is correct in the sense that it produces correct results for all specified test conditions (Ref. 16). It is conducted to determine whether a program complies with its specification under realistic operating conditions. This determination is made by evaluating both the code and the test results comprehensively to ensure that all functional, performance, and quality requirements are satisfied (Ref. 17). Satisfaction is governed by the criteria established by the user in the specification.

Test and evaluation can be performed by either development personnel or by an independent test team using a variety of approaches. Typically, development and test of microprograms progresses from the bottom up because of efficiency and performance considerations. The lowest level modules in the design are coded and unit-tested before the environment in which they are used is built because testing on the actual machine is impractical for most applications. Sophisticated microprogram development systems such as the ATMAC System (Ref. 18) illustrated in Figure 2 are developed to provide the environment for microprogram test. Driver programs are used to exercise the environment and provide the tests necessary to check out both the coded modules and the integrated package. Development continues from this point upward until all modules (i.e., closed sequences of microinstructions) are integrated together and checked out. Test and evaluation of the integrated package is conducted to confirm satisfaction of the specified requirements. Each of these requirements must be demonstrated.

Alternately, a top-down test approach can be utilized. In top-down implementation and test, the upper levels of a design that contain the overall control logic and data definitions at the microarchitectural level are mechanized first on a simulator or an emulator. Instead of using drivers for testing, smart stubs (Ref. 19) that simulate the consumption of critical control memory resources (i.e., both storage and timing) are used. Key algorithms (e.g., those used for instruction decode, etc.) are developed and tested early because of their potential impact on processor performance. When this upper level has satisfactorily finished testing, the process is continued, proceeding downward

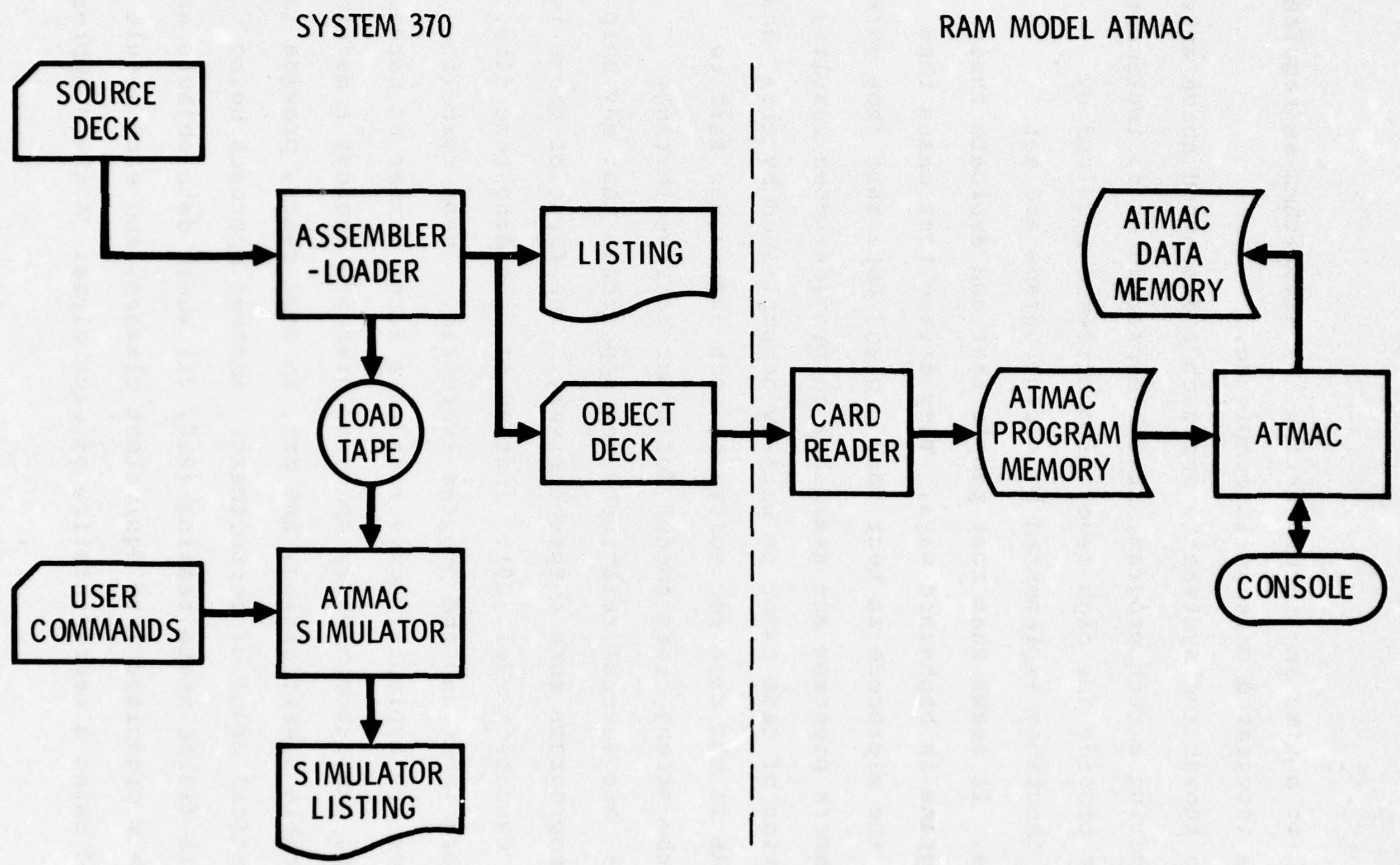


Figure 2. ATMAC System

one level at a time and incorporating key algorithms as required, until the integrated package is completed.

Even though the systematic approaches discussed above exist for conducting a test program, suitable procedures to implement them that provide the high level of confidence required by critical functions implemented by microprograms are not available. It seems that most people test and evaluate their microprograms in haphazard ways. They devise test cases that exercise the microcode as best they can and hope that they work. Few benchmark programs are available to provide known results, and creation of test cases is usually accomplished by trial and error. As is the case for software, such approaches fail to provide the strong basis needed for comprehensive testing.

There are several relatively new approaches that may help us test microprograms more comprehensively. The first of these is symbolic execution (Ref. 20). Instead of providing test data, symbols are input and the program exercised. These symbolic executions are typically equivalent to a large number of normal test runs, can automatically check the results against a machine-interpretable specification, and can, in some cases, provide for a mathematical proof of correctness. Another approach being pursued is finite state testing (Ref. 21) where data objects are effectively organized into equivalent classes, and each module is exercised using a representative of each class. A third approach that could be applicable in the future is attempting to develop a theory of program test that makes it possible to automatically generate both test data (Ref. 22) and a sufficient number of test cases to ensure that a program is thoroughly tested (Ref. 23). A final approach that could, if pursued, provide some near-term benefits seeks to create automated tools and software evaluation systems for use as aids during test and evaluation (Ref. 12).

Software evaluation systems (Ref. 10) deal with topological checkout of the structure of the code. Their objective is to increase the reliability of the module under test by uncovering logic and sometimes data errors. Their use to augment functional testing to ensure that (a) each source statement has been executed at least once, (b) each branch path has been executed at least once, and (c) each call has been executed at least once may be warranted if problems involving synchronization of highly parallel, logical micro-operations, and high instrumentation overhead are overcome.

A variety of other tools used for software testing could prove valuable to those involved in the test and evaluation of microprograms. Some of the more obvious of these include comparators used to identify changes in the source code, editors used to scan the source code and check relationships between sections of code, instruction traces used to create timed records of significant events based on data collected during microprogram execution, interface checkers used to check the range and limits of variables, map programs used to provide location and/or size data about selected portions of control storage, timing analyzers used to provide relative microinstruction execution timing data, and test-result processors used to perform output data reduction, formatting, and printing. Surveys describing these and other test tools are available (Ref. 24). As we stated in the previous section, an integrated system of tools designed to cope with the problems unique to microprogram development and verification and validation needs to be developed. Such a system would include tools for test and evaluation and an intelligent means for their selection and manipulation. Until such systems are implemented, we recommend that functional testing strategies should include testing using data-sensitive patterns, examining to ensure that each branch has been taken at least once, and checking to see that all null or "don't-care" conditions have been exercised.

C. PROGRAM PROVING

Progress has been made in developing systems that verify the correctness of computer programs with respect to formally defined specifications of the mathematical intent of the program (Refs. 25,26). The mathematical basis for the majority of these systems stems from Floyd's method of inductive assertions (Ref. 27). The formalization used for proofs arises from predicate calculus. Proofs of correctness are defined in terms of predicates that assert some relationship among the values of the variables of the program. Typically, the assertion at the beginning of a program is the relationship specified to exist upon entry, and the assertion at the end is the relationship that is required to exist upon exit. For each assertion it is proven that the truth of all other assertions implies the truth of this assertion (the method of induction).

Although progress has been made, practical application of program proving is conceded to be a long way off if realistic programs of reasonable size are to be handled (Ref. 28). According to Boyer (Ref. 29), the main obstacles to formally verifying realistic programs are:

a. The necessity (in most systems) for inventing intermediate assertions and the difficulty experienced by the programmer-verifier in coming up with these assertions

b. The difficulty in framing input and output assertions that adequately express the program's intent

c. The lack of sufficiently rich assertion languages in which to express these input/output assertions and intermediate assertions for programs covering a wide variety of data types and functional primitives

d. Technical limitations in present-day theorem-proving techniques (inadequate speed and large data storage requirements)

e. An apparent need for considerable programmer intervention in the theorem-proving process

After considering these limitations and the fact that even after a proof has been conducted, you cannot be sure that your program will run as intended on a given machine, it is no wonder that enthusiasm over the approach has diminished.

Let's determine if and how we can use the proof approach to verify the correctness of a microprogram giving ample consideration to the limitations of the technology. We shall assess the difficulty in using the technology for the following two purposes: establishing the correctness a priori of an algorithm that will be mechanized by a microprogram, and certifying the correctness of the microcode developed to implement an algorithm a posteriori.

The use of correspondence proofs to demonstrate the equivalence of a precise definition of an abstract machine at both the microinstruction and software architectural levels has been accomplished recently by Birman (Ref. 30). He carried out such a correspondence proof for the microprogram of a hypothetical stack machine (the S machine) to demonstrate the correctness of an emulation (Ref. 31). Leeman (Ref. 32) has expanded upon Birman's work to provide formal techniques for partitioning proofs of architectural equivalence. He recently described the interactive support system and the definition language (a subset of the Vienna Definition Language augmented with APL operators) used in the conduct of his investigations

(Ref. 33). This concept of equivalence is very useful because it allows us to establish the correctness of critical microprograms with respect to their strong equivalence prior to their implementation. Establishment of correctness a priori for critical functions (i.e., instruction semantics, recovery, etc.) is of extreme importance because these algorithms form the basis of the design and operation of the machine at the software level. Because these microprograms are typically small in size, are relatively well specified, are too complex for human comprehension, and are defined by skilled professionals familiar with the intricacies of the machine design, many of the disadvantages related to using proofs can be diminished when establishing microprogram correctness in the sense that the program solves the problem it claimed to solve. The costs and tedium associated with developing proofs of correctness for microprograms may be well justified by the fact that, in many situations, there may be no alternative other than trial and error and replacement of an entire read-only memory. Admittedly, major problems still exist in applying the technology to microprogram algorithm design and in addressing significant new technical issues specific to microprogram development. Some of these issues include developing a sufficiently rich definition language for establishing equivalence and specifying algorithm design, recognizing the different characteristics of microprograms as used to support different architectures, synchronizing simultaneous events occurring during the same machine state, and providing improved methods of generating simulations of the specifications as part of the microprogram synthesis process. We have used and found the APL language (Ref. 34) to be a useful microprogram design tool and recommend the user enhance it directly to support conduct of proofs during the analysis and design phases.

Proofs that certify the correctness of microcode a posteriori depend upon lemmas about components of the program and upon assertions that describe input-to-output transformations. Specification of these lemmas and assertions is extremely difficult when working at the microinstruction level because the program does not decompose easily into simple subschemas. When we add the complexity introduced by parallelism and the termination problem, we can conclude that it does not seem practical today to create meaningful proofs for microcode. We are not precluding development of new methods that will make proofs practical in the future, rather we are assessing the state of the technology to determine whether they can now be accomplished. We feel that when better languages and more disciplined techniques are used to produce microcode, proof techniques could be adapted for use selectively for critical modules during the unit checkout phase.

D. SIMULATION/EMULATION

Small computers provide the user with little visibility into the details of code execution, limited debug and test assistance, simple instruction sets, and insufficient peripherals. Interpretive Computer Simulators (ICSs) are the principal tool used to overcome these difficulties (Ref. 24). An ICS is a computer program that simulates the execution characteristics of a target computer using a sequence of instructions of some host computer. In simulating the target computer, the ICS provides bit-for-bit fidelity at the register transfer level with the results that would have been produced by the target computer following the same operations and initial conditions. These simulators can run from real time for the most optimistic cases to 6000 times real time for the most pessimistic cases. They are used because they permit full control over inputs and computer characteristics; they allow full-test repeatability; they permit use of sophisticated host computer diagnostics; and they are equipped to observe, monitor, collect data, and check for errors while the operational code is executing in its simulated environment (Ref. 35). Recently, an ICS has been developed that can be used to check out operational software and microcode at different levels of architectural data for a microprogrammed fault-tolerant computer (Ref. 36). The characteristics of the target computer an ICS typically simulates are displayed in Table 1.

Recent experience has shown that an Emulator-Based ICS (EBICS) is a viable alternative to an ICS. An EBICS implements those portions of an ICS that require high processing overhead (e.g., instruction emulation, trap processing, etc.) in the control store of a microprogrammable minicomputer in order to improve performance. For example, it has been reported than an emulation of the Trident Basic Processor executed over 100 times faster than a simulation of that machine written in FORTRAN running on a CDC 6400 (Ref. 37). As a second example, McClean (Ref. 38) reported than an emulator of the Univac Weapon System Controller ran with a 400 percent improvement over a CDC 6600-based ICS.

Emulators offer other advantages that may prove useful in verification and validation of microcode. They allow one to execute software and microprograms at different machine architectural levels with limited degradation in performance. They allow one to insert firmware probes and debug tools to access useful microprogram performance data (e.g., a dump of emulated writable control store could help optimization). They provide an environment where responsible allocation decisions and rational hardware/software/firmware tradeoffs can be made before the data system design is finalized. Just how one exploits these

TABLE 1

TYPICAL SIMULATED CHARACTERISTICS
1. INSTRUCTION SET
2. MEMORY SIZE
3. MEMORY WORD SIZE
4. MEMORY CHARACTERISTICS
• PAGING
• INTERLEAVING
5. CLOCK
6. REGISTERS
• INDEX
• GENERAL PURPOSE
• SPECIAL PURPOSE
7. INTERRUPTS
8. INPUT/OUTPUT
• SERIAL
• PARALLEL
• DISCRETE
9. SPECIAL HARDWARE FEATURES

features is a question that needs answering. Hopefully, current investigations being undertaken at several government agencies will provide the answer in the near future.

E. GRAPH-THEORETIC

The analysis of program structure is an important part of the validation process. It is important because of the relationships that exist between program structure and program correctness (Ref. 39), program complexity (Ref. 40), and test-case generation (Ref. 41). It has been shown that graph theory can be used effectively to model a program's structure in order to provide insight into the problems of correctness without regard to the level of detail under consideration (Ref. 42). We shall review the work accomplished to date and determine if it can be made applicable to microprograms.

A graph is a mathematical model of a program in which nodes represent program elements and edges represent lines of program flow. Program elements may be either single statements or groups of statements making up a program segment. Graphs have properties such as symmetry, reflexiveness, and completeness (Ref. 43) which can be exploited to make the mathematics more tractable. A graph is symmetric if every node satisfies the following condition: existence of an edge from node (a) to node (b) implies an edge directed from node (b) to node (a). A graph is reflexive if every node has a loop on itself. A graph is complete if every pair of nodes is connected in at least one direction. A directed graph or digraph is an irreflexive relation (Ref. 44) consisting of a set of nodes and edges having no parallel edges and loops. Acyclic directed graphs form a special class of digraphs where no two nodes are mutually reachable (Ref. 45).

There has been considerable progress made in applying graph theory to the solution of diverse computational problems. Russell (Ref. 46) has used graph models as the basis of his studies relating program structure to space and time requirements. Petri nets (Ref. 47) have been used to represent systems in which both static and dynamic conditions can exist simultaneously. Fosdick (Ref. 48) uses a graph model employing a depth-first search procedure to trace the pattern of references to a given variable along all possible paths in a time period that is linearly proportional to the number of edges in the program flowgraph. Ramamoorthy (Refs. 49,50) determines optimal flow through a program by generating a connectivity matrix to represent direct transfers in a program and extrapolating them using a reachability matrix. He has introduced algorithms to determine unessential nodes, loops in data flow, and parallel processing stream recognition. Kleir (Ref. 51) has used connectivity matrices to study optimization strategies for microprograms. Lipow (Ref. 23) uses graph models and an extension of Dilworth's theorem on partially ordered sets to determine the number of test cases needed to yield a given level of test thoroughness. A significant amount of other work using graph theory as its unifying approach appears in the literature.

It seems that graph theory can be used effectively to model a virtual vertical machine microprogrammed as a set of states (nodes), each of which is connected using state transitions (directed edges). Such a representation would enable us to decompose a microprogram into its primitive operations and verify that stated ordering rules for sequencing data flow and transforming machine states are obeyed as the program executes. Verification can be accomplished by testing a directed graph model representing the program for flow patterns and state transitions using symbolic or regular test data. Because a graph model directly corresponds with a flow chart of the program, new flow-charting techniques can be used to simplify the verification process. Figure 3 displays a flow chart of a microprogram drawn using a modified Nassi-Shneiderman charting technique (Refs. 52,53). Notice that each horizontal line represents a state vector containing several parallel, non-cooperative micro-operations and that their sequencing represents state transitions. Also, notice that null states are clearly marked and decision logic is clearly visible using this flow-charting technique. The approaches discussed probably are not usable for

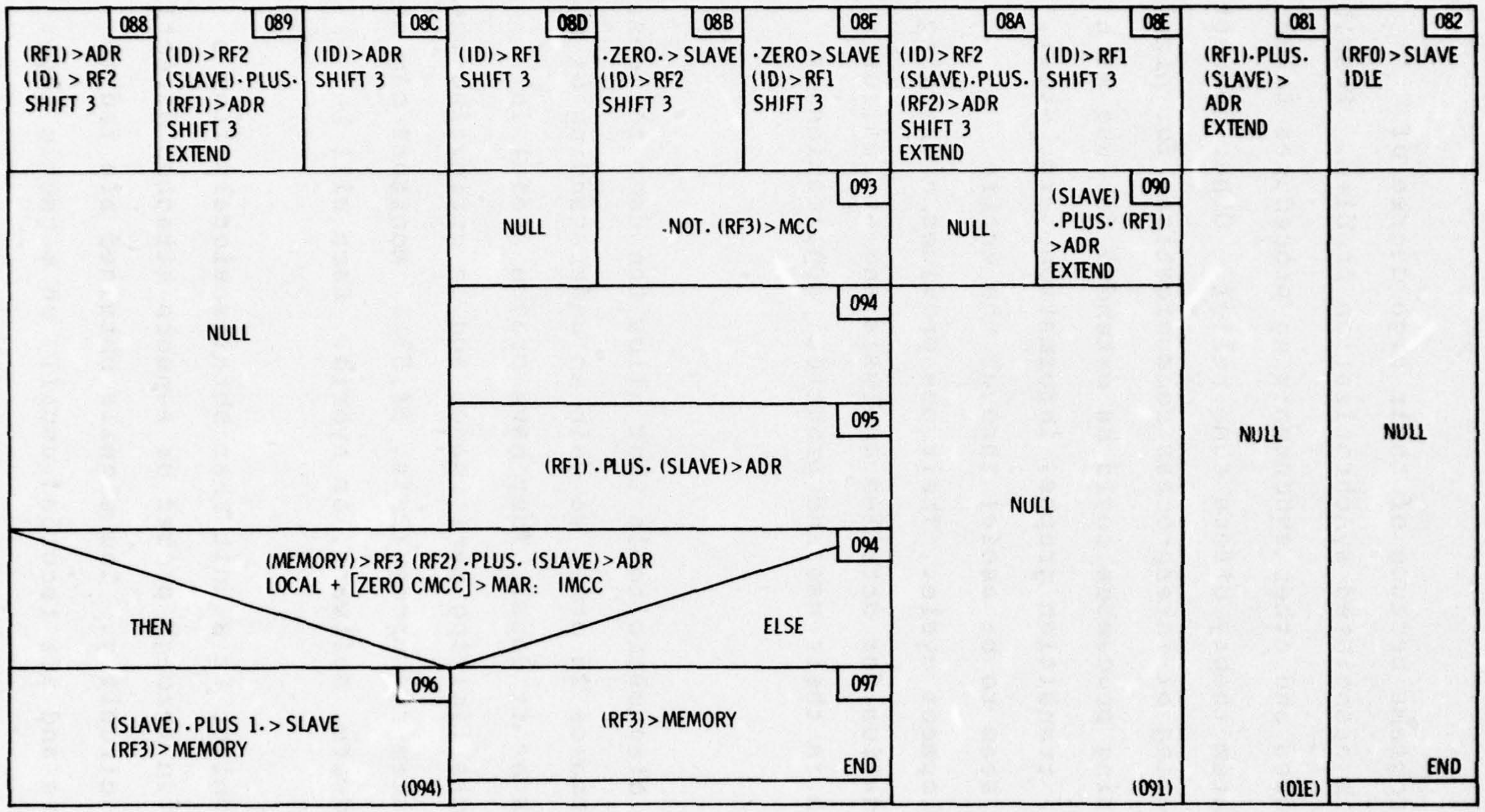


Figure 3. Nassi-Schneiderman Microprogram Flow Chart

horizontal microprograms because of their high degree of parallelism and its associated synchronization problem. Possibly the use of semaphores and other synchronizing procedures taken from operating system theory offers some relief. Other concepts developed for modeling of interprocess communications for highly parallel, cooperating processes could be extended for use with a graph model (i.e., transition graphs, information sets, etc.).

Graph models seem to be useful through the entire microprogram development cycle. Their use provides a unifying mathematical foundation for detailed analysis and evaluation. Continued research in their use and practical application is highly encouraged.

F. MONITORS

Monitors are diagnostic tools that allow the user to measure data system performance in order to gain an understanding of why his system behaves as it does. They have proven useful in locating bottlenecks limiting performance and in optimizing the utilization of system resources (Refs. 54,55). Monitors can be classified as hardware, software, or hybrid. Each will be discussed in turn.

A hardware monitor is a unit that obtains signals from a host computer system through probes or sensors attached directly to the computer's circuitry. The signals obtained are fed to counters and timers and are recorded usually on magnetic tape.

These data are reduced to provide information about CPU utilization, I/O utilization, peripheral utilization, degree-of-CPU-I/O-overlap, operation code usage, etc. The advantages of hardware monitors include that they do not affect or degrade the operation of the system, they can gather data on peripheral activity not reflected in the CPU, and they can obtain statistics (e.g., op code usage, etc.) without adding overhead. Their disadvantages include their cost, their inability to relate hardware utilization to the software in operation at the same time, and their inability to access software tables and gather queue, pointer, and certain program data (Ref. 56). Available hardware monitors are listed in Table 2 (Ref. 57). Their use in evaluating performance and optimizing microprograms is limited because they cannot keep up with the speed with which microcode executes within control store. Even the available minicomputer-based hardware monitors such as the DYNAPROBE 8000 are too slow. A hardware-monitoring approach with the speed to conceivably capture wanted data without affecting the operation of the host computer has been proposed by Murphy (Ref. 58). His System Logic and Usage Recorder utilizes an associative memory (i.e., content-addressable memory) to collect data via a special interface that detects and transmits signals when the channel status and internal computer conditions have changed. Adapting this technique for performance measurement of microprograms as an

TABLE 2

AVAILABLE HARDWARE MONITORS		
PRODUCT NAME	SYSTEMS	VENDOR
1. THE CAPACITY METER	IBM 360, 370	CRU
2. DSP-1	IBM 360, 370 HIS 6000 UNIVAC 1100 DEC 10	INT'L MARKETING AND CONSULTING LTD
3. DYNA-MYTE, DYNAPROBE 7900, 8000	ANY	COMRESS, DIV OF COMTEN
4. MS SERIES	ANY	TESDATA
5. TRICORDER	ANY	INT'L MARKETING AND CONSULTING LTD

adjunct to verification and validation is being pursued by the Space and Missile Systems Organization (SAMSO) of the U. S. Air Force with Aerospace Corporation support.

A software monitor is a computer program that provides detailed statistics about system performance. Because software monitors reside in memory, they have access to all the tables the system maintains. Therefore, they can easily examine such items as core usage, queue lengths, and individual program operation. They can be used continuously (full time) or at some rate less than 100 percent of the operation (sampling). The data they collect can be analyzed on-line or recorded for later reduction. Their major disadvantage is that they incur additional overhead in execution time and memory. This additional overhead distorts the system and resulting statistics. Some available software monitors are listed in Table 3 (Ref. 57). Creation of similar control-memory-resident monitors that capture data useful for performance evaluation and optimization is feasible but may not be warranted if overhead is excessive. Reduction of overhead by using nanolevel firmware probes to monitor performance at the microlevel seems plausible and is also being pursued by SAMSO (see Ref. 3 for a discussion of nanoprogramming and microprogramming and Ref. 59 for a description of a computer built using the concept).

TABLE 3

AVAILABLE SOFTWARE MONITORS

PRODUCT NAME	SYSTEMS	VENDOR
1. ALERT	IBM 360/50 UP 370/145 UP OS, VS	COMRESS DIV OF COMTEN
2. COPS	IBM 360, 370 OS/MVT/HASP, VS2	GRUMMAN DATA SYSTEMS
3. CUE	IBM 360, 370 OS, VS	BOOLE AND BABBAGE
4. DAPS	IBM 360, 370 OS, VS	COMPUTER MEAS-UREMENT RESOURCES
5. DSO	IBM 360, 370 OS, VS	BOOLE AND BABBAGE
6. DOS/RS	IBM 360, 370 DOS	DEARBORN COM-PUTER LEASING
7. DYNAMIC DISPLAY MONITOR	IBM 360, 370 OS	A.O. SMITH CORP
8. FAST/MASTER	CDC 3000	ADVANCED COMPUTER TECHNIQUES
9. LEAP	IBM 360, 370 OS, VS	DATA SERVICES CORP
10. OS MONITOR	IBM 360, 370 OS	LINCOLN LAND SOFT-WARE SYSTEMS
11. PMF	IBM 370 VM	STANDARD DATA CORP
12. QCM	IBM 360, 370 OS, VS	DUQUESNE SYSTEMS
13. RUM	IBM 360, 370 OS, VS	COMPUTER MEASUREMENT RESOURCES
14. SET	IBM 360, 370 DOS	DEARBORN COMPUTER LEASING
15. SLACMON	IBM 360, 370 OS	COSMIC
16. SUPERMON	IBM 360, 370 OS/MVT	COSMIC
17. SURVEY	IBM 360, 370 OS/ MVT/HASP, VS2	GRUMMAN DATA SYSTEMS

Hybrid monitors use both hardware and software measurement tools to gather pertinent data. An approach that seems feasible for evaluating microprogram performance follows:

a. Use a hardware monitor to sense and count event occurrences at the architectural level of the system.

b. Dynamically initiate, based on hardware monitor event occurrences, an associative memory data recorder to sense and count firmware event occurrences.

c. Dynamically initiate software monitoring strategies (i.e., use different speed probes and different sampling frequencies) as a result of the occurrence of selected hardware monitor or associative memory data recorder events.

d. Relate performance data gathered to the program (i.e., software and/or microprogram) in operation at this time.

e. Format and record the data recorded for future reduction and evaluation. Graphical techniques (e.g., Kiviat Graphs and dynamic time traces) are recommended for presenting the data because they are easily understood.

IV. CONCLUSIONS

Although still in their infancy, it seems that techniques for improved microprogram verification and validation are evolving along with a discipline for their application. Rapid advances will be made in their development by selectively adapting the best of existing software approaches for use in this application area. Unfortunately, such approaches will not suffice to provide assurance of satisfactory performance of those microprograms that implement critical system functions (e.g., recovery algorithms, firmware executives, etc.). The stringent requirements posed by these functions and their high cost of failure require better reliability and operability than are currently available for software. It has been established that there is no known way to comprehensively test a computer program of reasonable size and complexity (Ref. 60). Comprehensive verification and validation of microprograms is an essential prerequisite for the success of future systems employed by both the military and industrial sectors (e.g., failure of a manned satellite, a nuclear weapon, or a nuclear power plant could be catastrophic). Therefore, the challenge is raised, and new and innovative approaches must be devised. Hopefully, explorations into this problem area will provide the discoveries needed to effectively meet this challenge. Until then, a lot of hard work and perseverance will be required.

REFERENCES

1. S. S. Husson, Microprogramming--Principles and Practices, Prentice-Hall, Englewood Cliffs, NJ (1970).
2. A. Opler, "Fourth Generation Software," Datamation 13 (1) (1967).
3. G. F. Casaglia, "Special Feature: Nanoprogramming versus Microprogramming," Computer (January 1976), p. 54.
4. D. J. Reifer, Computer Program Verification/Validation/Certification, TOR-0074(4112)-5, Aerospace Corporation, El Segundo, CA (30 May 1974).
5. W. C. Hetzel, Program Test Methods, Prentice-Hall, Englewood Cliffs, NJ (1973), p. 8.
6. J. H. Gilder, "All About Microcomputers," Computer Decisions (December 1975), p. 44.
7. C. V. Ramamoorthy and M. Tsuchiya, "A High Level Language for Horizontal Microprogramming," IEEE Transactions on Computers C-23 (August 1974), pp. 791-801.
8. T. G. Rauscher and A. K. Agrawala, "On the Specification of Syntax and Semantics of Horizontal Microprogramming Languages," Proceedings of the ACM National Conference (1973).

9. P. W. Mallett and T. G. Lewis, "Considerations for Implementing a High Level Microprogramming Language Translation System," Computer (August 1975), pp. 40-52.

10. C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems," IEEE Transactions on Software Engineering SE-1 (1) (March 1975), pp. 46-58.

11. D. K. Kosy, Approaches to Improved Program Validation Through Programming Language Design, P-4865, RAND Corporation, Santa Monica, CA (July 1972).

12. D. J. Reifer, "Automated Aids for Reliable Software," Proceedings of the International Conference on Reliable Software, IEEE Cat. No. 75CH0940-7CSR (April 1975), pp. 131-142.

13. W. H. Joseph, An On-Line Debugger for OS/360 Assembly Language Programs, RM-6027-ARPA, RAND Corporation, Santa Monica, CA (August 1969).

14. R. M. Balzer, "EXDAMS--Extendable Debugging and Monitoring System," Proceedings of the Spring Joint Computer Conference, AFIPS Press, Montvale, NJ (1969), pp. 567-580.

15. J. Green, A Problem in Man-Computer Communications, Ph.D. Dissertation, Harvard University (1969).

16. Joel D. Aron, The Program Development Process, Part 1, Addison-Wesley Publishing Company, Menlo Park, CA (1974), p. 35.

17. D. J. Reifer, "Toward Specifying Software Properties," to be presented in the Software Engineering Session, IFIPS Conference, Amagi-Tokyo, Japan (April 1976). Available from author.

18. RCA, ATMAC Software Development System, Presentation (December 1975).

19. D. J. Reifer, "The Smart Stub as a Software Management Tool," submitted for consideration by 1976 National Computer Conference. Available from author.

20. J. C. King, "A New Approach to Program Testing," Proceedings of the International Conference on Reliable Software, IEEE Cat. No. 75CHO940-7CSR (April 1975), pp. 228-233.

21. P. Henderson and P. Quarendon, Finite State Testing of Structured Programs, No. 58, University of Newcastle upon Tyne, United Kingdom (February 1974).

22. W. E. Howden and L. G. Stucki, Final Report: Methodology for Effective Test Case Selection, MDC G5301, McDonnell Douglas Astronautics Co., Huntington Beach, CA (January 1974).

23. M. Lipow, "Some Directed Graph Methods for Analyzing Computer Programs," Proceedings of Computer Science and Statistics: Eighth Annual Symposium on the Interface (1974), pp. 357-363.

24. D. J. Reifer, Interim Report on the Aids Inventory Project, TR-0075(5112)-8, Aerospace Corp. (16 July 1975).

25. D. I. Good, "Provable Programs and Processors," Proceedings of the National Computer Conference, AFIPS Press, Montvale, NJ (1974).

26. S. Igarashi, R. L. London, and D. C. Luckham, Automatic Program Verification I: A Logical Basis and Its Implementation, ISI/RR-73-11, University of Southern California, Information Sciences Institute, Marina del Rey, CA (May 1973).

27. R. W. Floyd, "Assigning Meanings of Programs," Proceedings of Symposia in Applied Mathematics 19, American Mathematical Society (1967).

28. B. Elspas, K. N. Levitt, R. J. Waldinger, and A. Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys 4 (2) (June 1972), pp. 97-147.

29. R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution," Proceedings of the International Conference on Reliable Software, IEEE Cat. No. 75CHO940-7CSR (April 1975), pp. 234-245.

30. A. Birman, "On Proving Correctness of Microprograms," IBM Journal of Research and Development 18 (3) (May 1974), pp. 250-266.

31. G. B. Leeman, Jr., "Some Problems in Certifying Microprograms," IEEE Transactions on Computers C-24 (5) (May 1975), pp. 545-553.

32. ______, W. C. Carter, and A. Birman, "Some Techniques for Microprogram Validation," Information Processing 74, North-Holland Publishing Co. (1974), pp. 76-80.

33. ______, "Microprogram Validation by Algebraic Simulation," Presentation, Workshop on Methods of Verification in Design Automation, Michigan State University (8-10 October 1975).

34. A. D. Falkoff and K. E. Iverson, "The Design of APL," IBM Journal of Research and Development (July 1973), pp. 324-334.

35. R. D. Hartwick, Test Techniques for Large-Scale Programs, Short Course Notes, Software Reliability, Engineering 819.59, University of California at Los Angeles (October 1974).

36. D. D. Burchby, L. W. Kern, and W. A. Sturm, "Specification of the Fault-Tolerant Spaceborne Computer (FTSC)," to be published in Proceedings of the 1976 International Symposium on Fault-Tolerant Computing, IEEE (June 1976).

37. C. W. Flink, "A Microprogrammed Environment for a Software Development System," _Proceedings COMPCON Fall 75_, IEEE (September 1975), p. 46.

38. R. K. McClean and B. Press, "Improved Techniques for Reliable Software Using Microprogrammed Diagnostic Emulation," _1975 International Federation of Automatic Control Congress Proceedings 4_.

39. B. H. Liskov, "A Design Methodology for Reliable Software Systems," _Proceedings of the 1972 Fall Joint Computer Conference 41_, AFIPS Press, Montvale, NJ (1972), pp. 191-199.

40. J. E. Sullivan, _Measuring the Complexity of Computer Software_, MTR-2648, MITRE Corporation, Bedford, MA (1973).

41. E. F. Miller, "Structurally Based Automatic Program Testing," _Proceedings, EASCON-74_, Washington, D.C. (7-9 October 1974).

42. M. R. Paige and E. F. Miller, _Methodology for Software Validation--A Survey of the Literature_, RM-1549, General Research Corp., Santa Barbara, CA (March 1972).

43. P. D. Stigall and O. Tasar, "A Review of Directed Graphs as Applied to Computers," _Computer_ (October 1974), pp. 39-46.

44. F. Harary, R. Z. Norman, and D. Cartwright, _Structural Models: An Introduction to the Theory of Directed Graphs_, John Wiley and Sons, New York, NY (1965).

45. J. L. Bear, "A Survey of Some Theoretical Aspects of Multiprocessing," ACM Computing Surveys 5 (1) (March 1973).

46. E. C. Russell, Automatic Program Analysis, 69-12, University of California at Los Angeles (March 1969).

47. R. E. Miller, "A Comparison of Some Theoretical Models of Parallel Computation," IEEE Transactions on Computers C-22 (August 1973), pp. 710-717.

48. L. D. Fosdick and L. J. Osterweil, "DAVE--A FORTRAN Analysis System," Proceedings of Computer Science and Statistics: Eighth Annual Symposium on the Interface (1974), pp. 329-335.

49. C. V. Ramamoorthy, "Connectivity Considerations of Graphs Representing Discrete Sequential Systems," IEEE Transactions on Computers (October 1965), pp. 724-727.

50. ______, "Analysis of Graphs by Connectivity Considerations," Journal of the ACM 13 (2) (April 1966), pp. 211-222.

51. R. L. Kleir and C. V. Ramamoorthy, "Optimization Strategies for Microprograms," IEEE Transactions on Computers C-20 (July 1971), pp. 783-794.

52. I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming," SIGPLAN Notices (August 1973), pp. 12-26.

53. System Development Corporation, Microcode Development and Verification, Presentation (24 October 1975).

54. T. E. Bell, Computer Performance Analysis: Measurement Objectives and Tools, R-584-NASA/PR, RAND Corp., Santa Monica, CA (February 1971).

55. M. E. Drummond, "A Perspective on System Performance Evaluation," IBM Systems Journal 4 (1969).

56. J. Gertler, et al., Computer Systems Performance Measurement Techniques, DOT-TSC-FAA-71-23, Transportation Systems Center, Cambridge, MA (July 1971).

57. P. C. Howard, "Monitors and Merriment," Computer Decisions (September 1975), p. 41.

58. R. W. Murphy, "The System Logic and Usage Recorder," Proceedings of the 1969 Fall Joint Computer Conference, AFIPS Press, Montvale, NJ (1969), pp. 219-230.

59. QM-1, Hardware Level User's Manual, Second Edition, Revision 3, Nanodata Corp., Williamsville, NY (March 1975).

60. B. W. Boehm, Software and Its Impact: A Quantitative Assessment, P-4947, RAND Corporation, Santa Monica, CA (December 1972).